TÉCNICO LISBOA

Instituto Superior Técnico

Parallel and Heterogeneous Computing Systems

1º Semester 2021/2022

3º Report
OpenMP

Group 7

Nome	Número
Pedro Silva	87103
André Pereira	90016
Filipe Santos	90068

Day of the week: Monday, from 11:30 AM to 2:30 PM.

Professor: Aleksandar Ilic

1 PageRank

1.1 Problem description

Pagerank is an algorithm that ranks the importance of each vertex in a graph, based on how the edges are distributed along the graph. This algorithm, when applied sequentially, can take quite a bit of time, since it is needed to iterative compute the rank of each vertex until the absolute of the sum of the scores of all vertexes converges to less than a convergence parameter previously defined. When graphs are in the orders of billion vertexes and even more edges, like the ones used in real world applications, the execution time can start to drag.

1.2 Performance optimizations

In order to achieve better timings, we implemented several optimizations and parallelized the code:

- Firstly, we created a vector called no_outgoing_nodes to store all the nodes that do not have any outgoing edges (when we are initializing the arrays used on the problem). This way, whenever it is needed to compute the influence these nodes have on others, we do not need to check all the nodes to find them time and time again. We also used the function $\_\_sync\_fetch\_and\_add$ provided by the teacher to store the counter on how many nodes exist that had no outgoing edges, which is sightly better than having to create a critical region to increment the counter.

- We noticed that the weight that all nodes without outgoing edges that is added to all the nodes of the graph is computes only once for each loop cycle. This way, we avoid having to calculate this term for each node of the graph, which would require to constantly check all the nodes in the graph, and only do this once. Besides, we only perform the sum in a parallel fashion, and then multiply the sum result with the damping and divide by the number of nodes only on the cumulative sum. This saves these two final operations from being repeated endlessly on the loop for, even though it forces us to insert a OMP single operation to perform the computation on the cumulative sum. We called this variable buffer_score

- We also use the solution vector to store the old scores of the algorithm. Since we update the values of old and new scores just before the end of each loop cycle and before the verification of the condition, we save memory by not creating an extra vector with the size of the vertexes of the graph and save a for loop through all the edges to copy from said, hypothetical, extra vector to the solution vector.

- We perform a OMP reduction operation of sum for both the buffer_score and the global_diff in each of the respective loops that compute them. This way, we avoid

race conditions on the calculation of these variables and utilize the optimized openMP primitive to achieve the correct results.

- We divided only the work of computing scores of the nodes into 10000 tasks and scheduled the workload dynamically between the threads. During testing, this proved to achieve the best timings to beat the reference timings. Less than 10000 tasks started to drag the program performance because the workload per task would be too big, and threads would remain idle while others were working, and if we did more tasks, the overhead of creating, getting and computing small tasks would also degrade the performance. Scheduling the work dynamically also proved to be better than statically, since the program cannot now *a priori* the best distribution of the tasks, and it was better to assign the tasks during runtime to achieve a better work distribution.

1.3 Performance hazards

- We created a parallel section that is created in each iteration of the loop that is gonna compute the scores of the vertexes in an iterative way. Instead of ”encapsulating” the loop in the parallel region to save some overhead of the constant creation and destruction associated with the parallel section, we used this solution since it removed some necessary synchronization points/barriers that were needed (for example, when resetting the values of the buffer_score and the global_diff in each iteration of the loop). When testing, this achieved slightly better results. However, the lower the convergence parameter, the more cycles would be needed in the loop, and, probably, it would yield better results the solution that would ”encapsulate” the while loop.

- There are a lot of implicit barriers in the end of each pragma for loop. These synchronization points are needed since we cannot move to the next execution point of the program without having all the operations of said loop completed, since some threads could touch or start to compute values that would need variables computed possibly by other threads, and if said threads have not finished their work before these computations are done, the algorithm execution could lead to wrong results.

- We also allocate 2 extra vectors to help perform operations on the algorithm: score_new and no_outgoing_nodes. Both of these are allocated (having the size of all the nodes in the graph), initialized and later freed. Even so, the benefits of having these two vectors outweights the associated necessary extra memory space and overheads.

1.4 Performance results

Table 1: Performance Results for PageRank

Graph	$Reference_{seconds}$	Threads	$Achieved_{seconds}$	Speedup	Thread Usage
livejournal	3.6	1	5.7894	1	100%
		2	3.5900	1,6127	80,635%
		4	3.1783	1,8215	45,5375%
		8	3.1458	1,8403	23,00375%
orkut	4.8	1	7.4340	1	100%
		2	4.0491	1,8360	91,8%
		4	3.0393	2,4460	61,15%
		8	2.9347	2,5332	31,665%
200m	24.3	1	32.9219	1	100%
		2	22.3977	1,4699	73,495%
		4	25.1689	1,3080	32,7%
		8	23.8839	1,3784	17,23%
500m	51.9	1	71.6177	1	100%
		2	51.5547	1,3892	69,46%
		4	52.9036	1,3537	33,8425%
		8	51.9993	1,3773	17,21625%

Ideally, we expect a speedup equal to the number of threads if the program was fully parallel. Theoretically, the program will always have overheads associated with threads creation and communication, as well as uneven workload balancing, making it impossible to achieve a speedup equal to the number of threads. Looking at the results, we can see that the speedup was not linear with the number of threads used. In this algorithm, the major problems that stops us from achieving better speedups is the workload balance, which is difficult to predict, since the graphs can have many different shapes, and different nodes within the same graph can have massive discrepancies in the workload of the score calculation, and the program being memory bound, since graphs with this amount of nodes require constant access to the memory to fetch data, making the limiting factor of solving this problem the memory access speed.

2 Top-down Breath First Search algorithm

2.1 Problem description

Breath First Search algorithm is a famous algorithm that computes the distance of a node v to a certain root node. This can be used to find the distance between two specific nodes or, in this case, to find out the distance of all the nodes to the root node (if they are connected).

2.2 Performance optimizations

In this problem, the base code was given and, therefore, the code was only parallelized.

- First we parallelize the initialization of the distances vector in the beginning of the algorithm. Afterwards we had two options. Either paralellize the entire while loop and use OMP singles on the serial instructions of the loop or only parallelize the for loop within the while loop. Both have shown similar performance so we ended up only parallelizing the inner loop. This means that the time taken in killing and spawning the threads in each loop iteration is the same time consumed by the OMP singles created for the serial part of the code if we put the outer loop inside the parallel region (since it also creates a barrier).
- In the inner loop that we parallelized, it isn't a perfect parallelization either. Since we want to add nodes to a vector, the index where we need to add the next node is a shared variable and, therefore, needs to be a instruction that each thread doesn't execute at the same time.

2.3 Performance hazards

The paralellization creates a few problems that we had to tackle.

- Firstly, in the inner loop that we parallelized doesn't lead to a perfect parallelization. Since we want to add nodes to a vector, the index where we need to add the next node is a shared variable and, therefore, needs to be a instruction that each thread doesn't execute at the same time. This leads to a slower program.
- Also, when the distance vector is compared, if two threads are looking at the same value, they can't both pass the condition. This also creates a critical region like the previous point, since the threads can only enter this instruction one at a time. This slows the program down a lot and, for that reason, we added another condition with the exact same statement but without the critical region. This means the program enters the critical region fewer times, because it happens only when it might pass the condition and not every time.

2.4 Performance results

Table 2: Performance Results for Top-down Breath First

Graph	$Reference_{seconds}$	Threads	$Achieved_{seconds}$	Speedup	Thread Usage
1000X1000	0.02	1	0.02	1	100%
		2	0.04	0.43	21,5%
		4	0.04	0.39	9,75%
		8	0.03	0.53	6,625%
livejournal	0.20	1	0.43	1	100%
		2	0.33	1.29	64,5%
		4	0.29	1.48	37%
		8	0.25	1.74	21,75%
orkut	0.23	1	0.37	1	100%
		2	0.38	1	50%
		4	0.32	1.17	29,25%
		8	0.28	1.32	16,5%
200m	2.29	1	3.54	1	100%
		2	2.95	1.20	60%
		4	2.89	1.22	30,5%
		8	2.56	1.38	17,25%
500m	5.13	1	7.84	1	100%
		2	5.87	1.33	66,5%
		4	5.99	1.31	32,75%
		8	5.25	1.49	18,625%

Just like before, the ideal would be a speedup equal to the number of threads running. That is not what happens for several reasons. The synchronization between threads slows down the program, the spawning and killing the threads also creates a small overhead. And, also, there is a serial part of the code that can't be parallelized. In this case there isn't much workload in-balance tested by creating tasks with different sizes scheduled dynamical and statically.

3 Bottom-up Breath First Search algorithm

3.1 Problem description

The bottom-up implementation of the breadth first search is essentially the same as the the top-down, but it iterates through the graph differently. Instead of using a frontier as the main iterable structure, it looks through each vertex in the graph to check if its parent

node belongs to the frontier and adds them to it in case they were not visited yet. This can be great when the current frontier is large and expanding, as the nodes analyzed will more frequently find themselves to be added to the next frontier, although in other cases it can be sub-par.

3.2 Performance optimizations

We started off by writing an implementation of a serial version of the bottom-up version of breadth first search algorithm as illustrated in the assignment description, which was then parallelized.

- As a first step of parallelization, we simply added OMP parallel for to parallelize the initialization of the vectors and the main for cycle in the step part of the algorithm, and using the $\_\_sync\_fetch\_and\_add$ function as in the top-down version to update the frontier the same way, avoiding any data races.
- After this, we realized that for this algorithm, it wasn't necessary to compare the incoming edges of the vertex with the outgoing edges of the frontier as long as we had the value of the distance, which is the same as the number of steps taken so far, since that could be compared to the distances of the nodes connected by the incoming edges instead.
- Looking at the implementation a third time, we also realized that having a frontier was therefore useless since we ended up not using it, but instead just an iteration value, and given that fact, the $\_\_sync\_fetch\_and\_add$ and frontier updates were removed in the bottom-up implementation, eliminating the synchronization, which was then replaced with an auxiliary variable that was reduced at the end to represent the size of the frontier still necessary to determine the algorithms finish.

3.3 Performance hazards

This implementation should average to a equally distributed work load, since it doesn't have synchronization inside the main for loop and the vertices are distributed among the threads, but it has a problem.

The for loop iterates over the entirety of the vertices for every single step. This leads to a complexity of O($\frac{V}{num\_threads}^2$), which wouldn't be necessary if the vertices were stored in a structure containing only the remaining vertices, so that each iteration of the for loop would only check non visited vertices, which we didn't manage to implement.

3.4 Performance results

Table 3: Performance Results for Bottom-up Breath First

Graph	$Reference_{seconds}$	Threads	$Achieved_{seconds}$	Speedup	Thread Usage
1000X1000	1.52	1	4.14	1	100%
		2	2.31	1.79	89,5%
		4	1.41	2.94	73,5%
		8	1.27	3.24	40,5%
livejournal	0.14	1	1.08	1	100%
		2	0.59	1.84	92%
		4	0.40	2.70	67,5%
		8	0.39	2.79	34,875%
orkut	0.15	1	1.29	1	100%
		2	0.67	1.93	96,5%
		4	0.38	3.36	84%
		8	0.33	3.87	48,375%
200m	1.98	1	7.53	1	100%
		2	4.63	1.63	81,5%
		4	4.58	1.64	41%
		8	4.44	1.70	21,25%
500m	19.67	1	35.19	1	100%
		2	26.99	1.30	65%
		4	27.31	1.29	32,25%
		8	26.32	1.34	16,75%

As it is somewhat expected, we can see that this implementation does not deal with higher numbers of vertices that well because of the aforementioned complexity, even with a evenly distributed load and minimal synchronization. It should also be noted that the worst complexity isn't the sole determiner of the relative performance, but also the graph density itself, since this would slow down implementation of bottom up that only iterates through not visited nodes more than the current one.

4 Hybrid Breath First Search algorithm

4.1 Problem description

In the hybrid implementation, we created an algorithm that takes into account both previous implementations of the bfs, and tries to utilize either one of these implementations between steps. To decide which implementation to use in each step, we take a look

at the amount of vertices in the frontier, and check if it is lower than a certain ratio of amount of nodes from the whole graph, meaning:

$$n_f < \frac{n}{\beta} = C_{BT} \tag{1}$$

where n_f is the vertices at the frontier, n is the total amount of vertices in the graph and β is a compensation factor. If n_f really was less than the other term of the equation, we used the top down implementation, and if it was not, we would use the bottom up implementation. Therefore, we would dynamically choose between these two based on the size of the frontier.
While testing, we found that $\beta = 4$ was a reasonable compensation factor to have, and it sufficed to beat the reference timings. The overhead of deciding the implementation is barely noticeable, since it is purely an if statement between two variables between each step.

4.2 Heuristics

Even though we did achieve to beat the reference timings for the hybrid implementations, we did a bit of research online because we believe that the condition above mentioned is not the best factor to purely decide which method to use in all the cases. What we hoped to find was that the amount of edges from the frontier nodes and the amount of edges from the unvisited nodes also had a weight while deciding, because the amount of edges that are necessary to iterate between the nodes should impact the performance of the algorithm, since both top down and bottom up have to iterate between the edges of the nodes they have to analyze. If we have a frontier with some nodes that have a lot of edges, meaning the frontier is dense, maybe it can be better to just check the unvisited nodes, that might be in a greater amount than the ones in the frontier, but can also have a low amount of edges to traverse. And we stumbled upon some heuristics online that did, in fact, helped the hybrid algorithm to pick. By using an extra equation besides the one we arrived at, namely:

$$m_f > \frac{m_u}{\alpha} = C_{TB} \tag{2}$$

where m_f is the edges adjacent to the frontier, m_u is the edges adjacent to the unvisited nodes and α is a compensation factor, it is possible to create a state machine that could control the decision making of the algorithm:

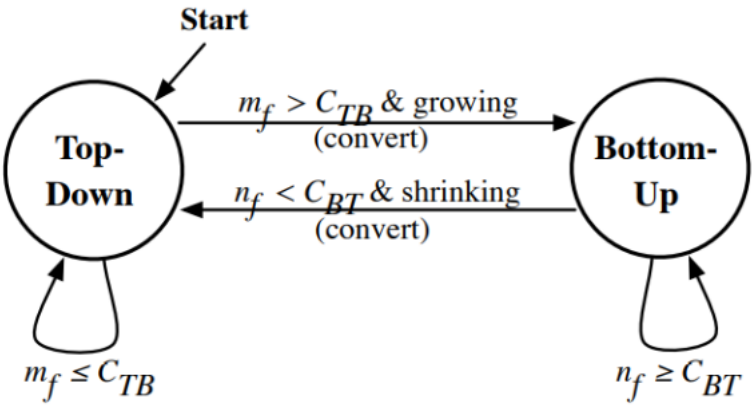

Figure 1: State machine that controls when to switch between bfs implementations

This technique would create a larger overhead than the one we used, since it would require to create a simple control unit for the hybrid implementation. Also, in the top down implementation, we would need to keep track of the amount of edges from both the frontier and the unvisited nodes, which would require an extra couple variables to work with and reduction steps on each iteration. Still, even with the larger overhead associated, it is very likely that this would result in a better program performance, since it should decide much better when to use each bfs implementation, and the performance gain from a better choice should largely over weigh the added overhead from this technique. Unfortunately, we did not have enough time to test this approach to the hybrid bfs. Besides, like stated before, we already achieved slightly better timings than the reference timings, so we could only analyze the technique.

4.3 Performance results

Table 4: Performance Results for Hybrid Breath First Search

Graph	$Reference_{seconds}$	Threads	$Achieved_{seconds}$	Speedup	Thread Usage
1000X1000	0.52	1	0.02	1	100%
		2	0.04	0.43	21,5%
		4	0.04	0.40	9,75%
		8	0.03	0.53	6,625%
livejournal	0.09	1	0.27	1	100%
		2	0.23	1.18	59%
		4	0.21	1.30	32,5%
		8	0.17	1.60	20%
orkut	0.05	1	0.05	1	100%
		2	0.11	0.45	22,5%
		4	0.10	0.48	12%
		8	0.08	0.58	7,25%
200m	1.22	1	1.86	1	100%
		2	1.48	1.26	63%
		4	1.31	1.42	35,5%
		8	1.17	1.58	19,75%
500m	3.42	1	3.82	1	100%
		2	3.37	1.13	56,5%
		4	2.83	1.35	33,75%
		8	2.57	1.49	18,625%

The only test that we were not able to surpass was the live journal graph. In this case, probably taking only into account the nodes at the frontier versus the nodes from the graph is not enough to take a wise decision between each of the bfs implementations will achieve a better performance for that given step.